Designing scientific SPARQL queries using autocompletion by snippets

Karima Rafes, Serge Abiteboul, Sarah Cohen-Boulakia, Bastien Rance

Designing scientific SPARQL queries using autocompletion by snippets

Karima Rafes
BorderCloud
LRI, U. Paris-Sud U. Paris-Saclay
Orsay, France
karima.rafes@bordercloud.com

Serge Abiteboul
Ecole Normale Superieure
Inria
Paris, France
serge.abiteboul@inria.fr

Sarah Cohen-Boulakia
LRI, U. Paris-Sud
U. Paris-Saclay
Orsay, France
cohen@lri.fr

Bastien Rance
APHP G. Pompidou
INSERM CRC
U. Paris Descartes Paris, France
bastien.rance@egp.aphp.fr

***Abstract*—SPARQL is the standard query language used to access RDF linked data sets available on the Web. However, designing a SPARQL query can be a tedious task, even for experienced users. This is often due to imperfect knowledge by the user of the ontologies involved in the query. To overcome this problem, a growing number of query editors offer autocompletion features. Such features are nevertheless limited and mostly focused on typo checking.**

In this context, our contribution is four-fold. First, we analyze several autocompletion features proposed by the main editors, highlighting the needs currently not taken into account while met by a user community we work with, scientists. Second, we introduce the first (to our knowledge) autocompletion approach able to consider snippets (fragments of SPARQL query) based on queries expressed by previous users, enriching the user experience. Third, we introduce a usable, open and concrete solution able to consider a large panel of SPARQL autocompletion features that we have implemented in an editor. Last but not least, we demonstrate the interest of our approach on real biomedical queries involving services offered by the Wikidata collaborative knowledge base.

***Index Terms*—Autocompletion, SPARQL, unsupervised learning, hierarchical clustering, bioinformatics, Wikidata.**

I. Introduction

The amount of linked data available on the Web is constantly increasing. Querying such data is a crucial need to exploit its wealth. When linked data is represented using the RDF standard, SPARQL is the most popular query language. However, writing a SPARQL query can be tedious, even for experienced users. Reasons are numerous and include the imperfect knowledge that users may have on the ontologies involved in the query or the need to follow a syntax rather complex to master.

To answer these problems, systems often propose interfaces hiding the SPARQL query language: the user is asked to input keywords, examples of expected results, or is proposed some canned queries. The drawback of these approaches is two-fold. First, users do not progress in learning the language although the volume of available data and the number of interesting applications constantly increase. Second, users are limited in their search not only by the data structure (ontology) supported by the interface but also and especially by the functions supported by the interface. In other words, users are limited in that they can ask only queries imagined by the system developers.

The solution we provide allows users to exploit the full power and expressiveness of the query language while guiding them in the design of queries, by proposing a large panel of *autocompletion* features.

The starting point of this paper is a two-year study that we conducted with users of SPARQL editors. This study was carried out as part of the use of the platform LinkedWiki by various types of users, novice to expert, from various disciplines (computer science, biology, chemistry, physics, human and social sciences, etc.) designing queries [1]. The goal of the platform LinkedWiki, also based on SPARQL, is to simplify enriching and reusing the Web of data with RDF semantics.

Some of the features we provide are already supported by other query editors. The study highlighted the need for a most useful feature for advanced users, namely *autocompletion by snippets*, that is exploiting fragments of queries made by previous users to complete the partial query already written by the user. This is the topic of the present paper. Such a feature is not considered by existing SPARQL editors [2]. A similar one has been considered for other query language editors [3–7] and more generally this has been studied in the context of data exploration [8].

In this paper, we introduce the first (to our knowledge) autocompletion by snippets for SPARQL. For this, we extract a representation of subpatterns from previous queries (under the form of "linegraphs") and construct a hierarchical clustering of such subpatterns to generate snippets that are most compatible with the portion of the query considered by the user. These snippets are then suggested to the user as autocompletions, making the editing process significantly simpler and quicker.

The remainder of the paper is organized as follows. Section II introduces the context of our study and provides a motivating example from the biomedical domain. Section III introduces needs for autocompletion collected from users; it also provides a panorama of the current autocompletion features supported by SPARQL editors. Section IV introduces SPARQLets-Finder, the first autocompletion approach based on snippets. Section V evaluates our approach both quantitatively, providing measures obtained on a large set of use cases and qualitatively, based on the queries of interest introduced in Section II.

II. Context

General context. Since 2015, we have been helping users to write, share and discover new SPARQL queries using two instances of the LinkedWiki platform [1]: one instance dedicated to scientists and students from the Paris-Saclay University[1] and another instance available to anyone[2]. We carried out a study of the user needs for autocompletion when writing queries, by collecting during the last two years the wishes of users, from novices to experts. More precisely, this experiment has been performed with two types of users: 103 master students with various scientific backgrounds (computer science, business administration, etc.) and various skills in SPARQL, and 60 professionals from industry and academics following training sessions in the *Center for Data Science* of Paris-Saclay University. Most users were scientists from various disciplines including astronomy, life sciences, and social sciences, with little knowledge in computer science.

The approach presented in this paper can be used for the completion of queries in arbitrary domains. The context of the Center for Data Science turned to be appropriate to capture this generality. To illustrate more precisely our purpose, we will consider in the following a use case from the biomedical domain.

Biomedical use case. The use case we consider takes place in the context of personalized medicine and more precisely in the study of the variability of the response of several anti-cancer drugs depending on patients (that is, depending on genetic mutations of, e.g., specific genes of patients, genetics of their tumors, etc.). Here, the knowledge regarding drug responses is scattered over hundreds of scientific articles indexed in the PubMed repository (a public library of millions of biomedical publications), many of which are referenced in Wikidata.

While designing a query to find relevant information, the user encounters various difficulties. To be able to find relevant data, a SPARQL-educated physician would need a deep knowledge of Wikidata underlying ontology, of the organization of the information, and more generally at the exhaustive content regarding drugs, gene-mutation relations and cross-references to PubMed.

The physician first searches for diseases that are cancers, and finds drugs used to treat cancer. Although this first (sub)query is relatively simple, the physician is required to know the structure of the Wikidata ontology (e.g., the `rdfs:subClassOf` type), mostly a deep knowledge of Wikidata's identifiers for drug, cancer (identified by `wd:Q12078` in Wikidata), the predicate *treats* (identified by `wdt:P2175`), etc.

Without any autocompletion feature it is unlikely that the physician will succeed in writing the appropriate query.

The next part of the search should provide gene variants associated with better response to a given drug, and references regarding the provenance of the information.

[1]See https://io.datascience-paris-saclay.fr (version 2. last version)
[2]See http://linkedwiki.com (version 1.)

Again, without the help of any autocompletion feature, such a query appears to be over-complicated, possibly beyond the competence of the physician.

Interestingly, it is very likely that portions of the query have already been asked by previous users, e.g., searching for the provenance (references) associated to pieces of information are likely to have been asked by previous users. Being suggested to such query portions would greatly simplify the task of the user.

More generally speaking, our experience with users is the following: when designing a SPARQL query, the common errors are, by order of importance, syntactic errors, unknown prefixes, and imperfect knowledge of the ontologies involved in the query. This last category of errors is made by all kinds of users, from novice to expert.

We next briefly describe the tools that we think essential to help the users to create a SPARQL query.

III. Autocompletion features analysis

Recall that *Linked Data* is (to summarize) a method of publishing structured data via Internet, such as data of Wikidata [9]. The data is represented as a graph. An arc of the graph is *a triple* "subject-predicate-object". A *SPARQL query* is based on graph patterns that are matched against this graph.

The SPARQL language names all its addresses using *IRI* (*Internationalized Resource Identifier*): subject, predicate, object, etc. In a query, users can write absolute IRIs (without abbreviation) but they often prefer to use relative IRIs where the prefix of the IRI is replaced by an abbreviation declared before the text of the query. These abbreviations are named *prefixes* (see Figure 1).

A *SPARQL service* is a service delivered by an *RDF database* through the SPARQL protocol [10], responding to queries over the database.

Autocompletion (in our context) is a feature of a SPARQL editor [2] which, when requested by the user, proposes different ways of completing the portion of the query written so far.

We next briefly present autocompletion features of which our users expressed the need. For each, we specify the kind of users requesting the feature (novice or expert).

Autocompletion using relative IRIs, via keywords.
This autocompletion of IRIs does not presuppose any prior knowledge of the SPARQL service and the ontologies it contains. The user selects keywords, in the language of her choice, to obtain a list of suggestions for relative IRIs. It is then enough to choose one suggestion so that the tool can insert it in the current request with the definition of the prefix. This type of feature is requested by both kinds of users.

Autocompletion by prefix declaration.
This autocompletion inserts declarations of prefixes not explicitly specified by the user into a query. It responds to a recurring need for users to reuse examples of queries available on the Web in a new context where prefixes

must be clarified. This feature is highly requested by all kinds of users.

Autocompletion by template.
A template provides users with an example of query structure to build faster a query. Generally, propositions of templates are designed in advance, manually, by experts, more particularly for novice users of the system.

Autocompletion by suggestion of snippets.
A *snippet* is a piece of reusable code proposed in order to complete the current query. It can be seen as the prediction of the future query fragment the user would write to complete his query. This feature is highly demanded by experienced users. It may also help novices designing their first queries and getting familiar with the language.

Autocompletion in editors. We next consider SPARQL editors discussed in [11] and several others. We limit our attention to the eight editors that include at least one autocompletion feature expected by our users, and that could be tested using a Web navigator.

Results are summarized in Table I.

The two most well-known editors are Flint SPARQL Editor [12] and YASGUI [11]. Flint proposes query templates. YASGUI aggregated several advanced features such as *autocompletion by prefix declaration* with Prefix.cc API [13]. Gosparqled [14], that builds on YASGUI, also provides autocompletion IRIs by keywords.

Another family of solutions comes from RDF databases providers proposing *single-point* editors, i.e., editors dedicated to a single SPARQL service. Such editors are often rather basic. One exception is iSPARQL [15] where autocompletion by template is provided (among many other functionalities).

Some other solutions are domain-based such as BioCarian [16] providing templates specifically tuned to queries on molecular biology.

Wikidata editor [17] not only offers templates but also *autocompletion using relative IRIs* via keywords on the wikidata database. Interestingly, Gosparqled [14] and Wikidata support research by keywords in a completely different way. The first does it via a SPARQL query with REGEX function that is often too greedy when database is very large eventhough but the function is available in any SPARQL service. The second supports a specific API to search for Wikidata IRI via keywords.

As for the feature *autocompletion by snippets*, we would like to mention LODatio+ [18], that is not an editor but a search engine able to find linked data sources relevant to query needs on triple patterns related to a specific combination of RDF types and/or properties. LODatio+ indexes different schemas in a given knowledge base and proposes to add or remove snippets of one triple pattern at a time in a query for searching data. This feature is the one we found that is the closest to our definition of *autocompletion by snippets*. However, it is very limited in that it only suggests one single triple at a time.

In conclusion, these editors support some autocompletion features, but in a limited way. None of them supports the four kinds of autocompletion expected by our users. Most importantly, completion by snippet is almost inexistent.

TABLE I
AUTOCOMPLETION FEATURES IN SPARQL EDITORS

Features	Flint Editor[a]	iSPARQL[b]	LODatio+[c]	BioCarian[d]	Gosparqled[e]	Wikidata Query[g]	YASGUI[h]	**LinkedWiki editor**[i]
Rel. IRI via keywords	-	-	-	-	+	+	-	+
Prefix declaration	-	-	-	-	+	-	+	+
Template	+	+	+	+	-	+	-	+
Snippets	-	-	+[j]	-	-	-	-	+

a http://bnb.data.bl.uk/flint-sparql Flint SPARQL Editor v1.0.4 [12]
b https://www.openlinksw.com/isparql OpenLink iSPARQL v2.9 [15]
c http://lodatio.informatik.uni-kiel.de LODatio+ (tested in may 2018) [18]
d http://www.biocarian.com BioCarian SPARQL editor (tested in may 2018) [16]
e http://scampi.github.io/gosparqled/ Gosparqled (tested in may 2018) [14]
g https://query.wikidata.org Wikidata Query (tested in may 2018) [17]
h http://doc.yasgui.org/ YASGUI v2.7.27 [11]
i **https://io.datascience-paris-saclay.fr/exampleInsertUpdate.php LinkedWiki platform v2.0.0**
j Only snippets of one triple pattern

In the paper, we introduce the LinkedWiki SPARQL editor that implements the full set of autocompletion techniques mentioned at the beginning of the section (the autocompletion using relative IRIs, via keywords, requires the existence of an adapted API that is provided by Wikidata).

The remainder of this paper is dedicated to the introduction of *SPARQLets-Finder*, the new module of the LinkedWiki editor, dedicated to the generation of snippets to users.

IV. ALGORITHM

In this section, we introduce the first (to our knowledge) snippet-based autocompletion solution. We first extract a representation of subpatterns present in previous queries (under the form of "linegraphs"). We then construct a hierarchical clustering of the queries subpatterns. Finally, we generate snippets that are most compatible with the portion of the query proposed by the user so far, and resemble previous queries. We start this section by giving an overview of the technique.

A. Overview

A SPARQL query is based on graph patterns that are matched against the knowledge graph. Figure 1 shows a query that contains a graph pattern. Queries are formed by combining smaller patterns. The smallest graph pattern, the ***basic graph pattern* (BGP)**, is a set of *triple patterns* involving both variables and constants. As a result, the graph structure representing a BGP will have vertices denoting variables and constants.

In the SPARQL queries written by users, we observed a lot of similarities between BGPs of queries of different topics. This lead us to the idea of suggesting snippets to the users. To discover these snippets, we use a hierarchical clustering based on a similarity distance between BGPs. The use of such a hierarchical clustering has been designed in the spirit of [19–22].

The effectiveness of this method is clearly conditioned to the quality of the similarity distance that is used. Defining

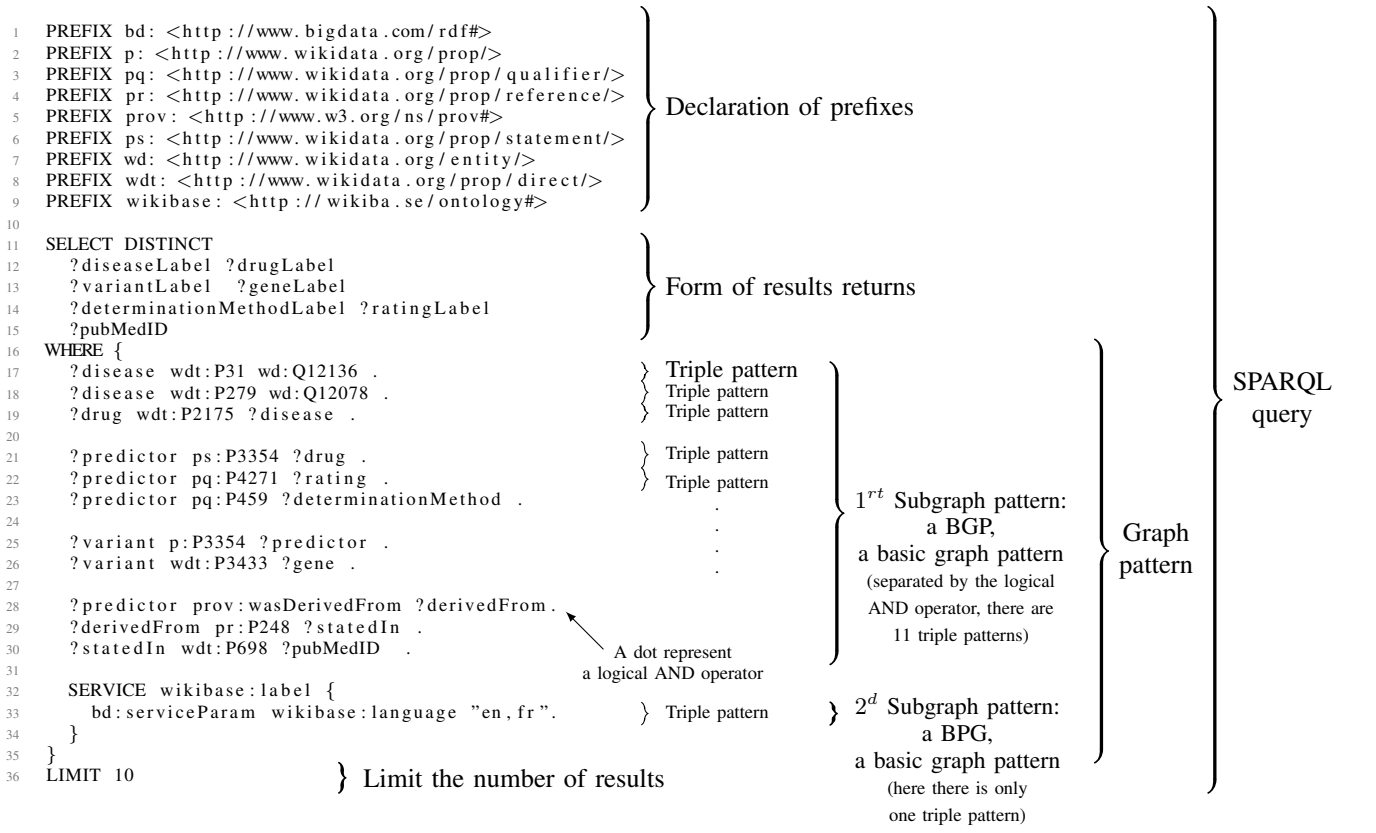


Fig. 1. A SPARQL query is based around basic graph pattern matching and contains sets of triple patterns. Here, we use our example of query where we removed all syntactic sugar to show clearly all the sequences of triple patterns in the graph pattern of the query.
See results of the query: https://io.datascience-paris-saclay.fr/query/Variants_associated_with_positive_treatment_response_to_anti-cancer_drug_with_provenance_information.

a similarity distance between graph patterns is not an easy task [23], which required from us lots of experiments.

Thus, in this paper, we describe two other contributions necessary to our snippet-based autocompletion solution. The first contribution is a method to highlight major features of BGP (Section IV-B) to compute a similarity distance used to compare patterns. The second is a BGP Hierarchical Clustering algorithm, namely BGPHC (Section IV-C), that we use to structure the common triple patterns found in queries over our knowledge base. This hierarchy turns out to be useful to find, given some BGPs present in the query proposed by a user, nearest BGPs already known. We believe that the method could be useful in other contexts as well, e.g., for building search engines able to retrieve queries.

These two contributions are the keys to implement the workflow (Section IV-D) used to find SPARQL snippets adapted to a partial query given as input.

In the next part, we explain how we compute a similarity distance between BGPs of queries.

B. BGP common structure and distance measure

Among approaches to compare patterns of queries [23], we see [24] as most interesting because it computes the similarity as a function of the structural pattern while remaining agnostic to the data context (i.e., agnostic to the ontology). As a result, it allows addressing all kinds of services and data. We decided to follow their technique by extracting BGP main features to find common patterns between queries.

First experiments with that technique showed that it was not satisfactory because of the uncontrolled uses of triple pattern predicates. The predicates of type "*is a*" (e.g., `rdf:type` or `wdt:P31`) used in a majority of queries create similarities that turn out to be meaningless. So, we adapted the method as follows.

Linegraphs. Starting from a BGP, we construct a graph, namely, a *linegraph*, that is a refined representation of the BGP and enables capturing meaningful similarities. More precisely the linegraph of a BGP G, noted $\mathcal{L}(G)$, is a graph such that each vertex of $\mathcal{L}(G)$ represents an arc of BGP; and two vertices of $\mathcal{L}(G)$ are adjacent if and only if their corresponding arcs share a common vertex of kind variable in the BGP G. Each vertex n of the linegraph is labeled as follows:

- case 1: when n represents a predicate in the BGP G of type "*is a*" which is associated to a constant object then the label of n is the concatenation of predicate label and the object label of the BGP G.
- case 2: when n represents any other predicate p in the BGP G then the label of n is the label of p.

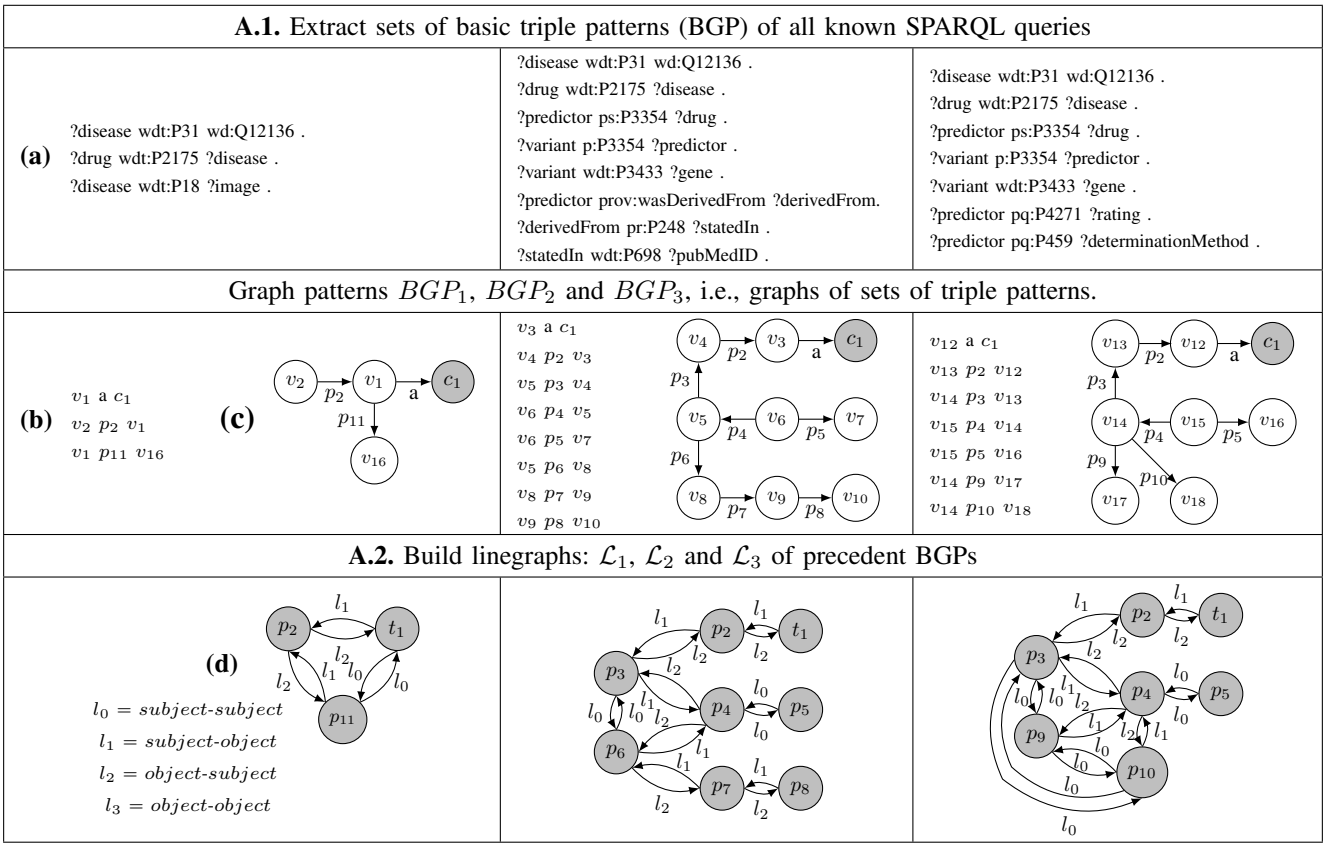


Fig. 2. Illustration of processes **A.1** and **A.2** to build the linegraphs $\mathcal{L}_1$, $\mathcal{L}_2$ and $\mathcal{L}_3$ of their BGP BGP_1, BGP_2 and BGP_3 extracted of queries q_1, q_2 and q_3.

As for the arcs, four types of arcs are considered to express the four possible joins between two triple patterns in a BGP sharing a common vertex (object or subject). We named these joins: *subject-subject*, *subject-object*, *object-subject* and *object-object*. An x-y (for x, y either subject or object, or both) join is used when the vertex common between the two triples is an "x" for the first triple pattern, and a "y" for the second. Arcs are then labeled as follows: l_0=*subject-subject*, l_1=*subject-object*, l_2=*object-subject*, l_3=*object-object*.

Illustration. We illustrate the concept of a BGP associated with a set of triple patterns and its associated linegraph in Figure 2. In the first column of Figure 2, (a) provides three triple patterns forming the basic graph pattern BGP_1 of the query q_1. In (b), the variable `?disease` in (a) has been associated with v_1, the predicate `wdt:P31` with a, the object `wd:Q12136` with c_1. v_2, v_{16}, p_2 and p_{11} are similarly constructed in graph BGP_1. (c) is the same basic graph pattern but represented graphically. From (c), the linegraph (d) can be built from the adjacencies between arcs: each predicate is transformed into a vertex (e.g., the predicate `wdt:P2175` is transformed into p_2). To deal with the specific case of predicates of type "*is a*" `wdt:P31` and its constant `wd:Q12136`, we create a new vertex t_1 that represents the association of `wdt:P31` with `wd:Q12136`. Moreover with the BGP of (c) in (d), the arc $p_2 \rightarrow t_1$ has the label l_2 because it replaces the vertex $v1$ which is the object in the triple pattern of p_2 and the subject in the triple pattern of c_1. Figure 2 provides other illustrations of linegraphs of BGPs extracted from queries.

Linegraph BGP distance. We now describe the method designed to capture the similarities between linegraphs. For each linegraph, we keep track of the set of linegraph arcs that it contains. Following [24], we designed a new distance measure between BGPs based on the Jaccard distance: LBGPD (for *Linegraph BGP Distance*) computes the similarity between the corresponding linegraphs depending on their common (i.e., shared) arcs.

We next describe how we compute the Hierarchical Clustering of BGPs that will provide an organized representation of the common linegraphs.

C. BGPHC: BGP Hierarchical Clustering

A divisive hierarchical fuzzy clustering. To compute snippets, we use a BGP Hierarchical Clustering, in short BGPHC, in the spirit of the divisive hierarchical fuzzy clustering of [22]. In the hierarchical clustering, each computed cluster is a set of linegraphs that share arcs. Each cluster is characterized by a set of common arcs. This defines the hierarchy. The clustering is *fuzzy* because a given linegraph may share arcs with different linegraphs, so may belong to several clusters. Since the number of clusters in the result

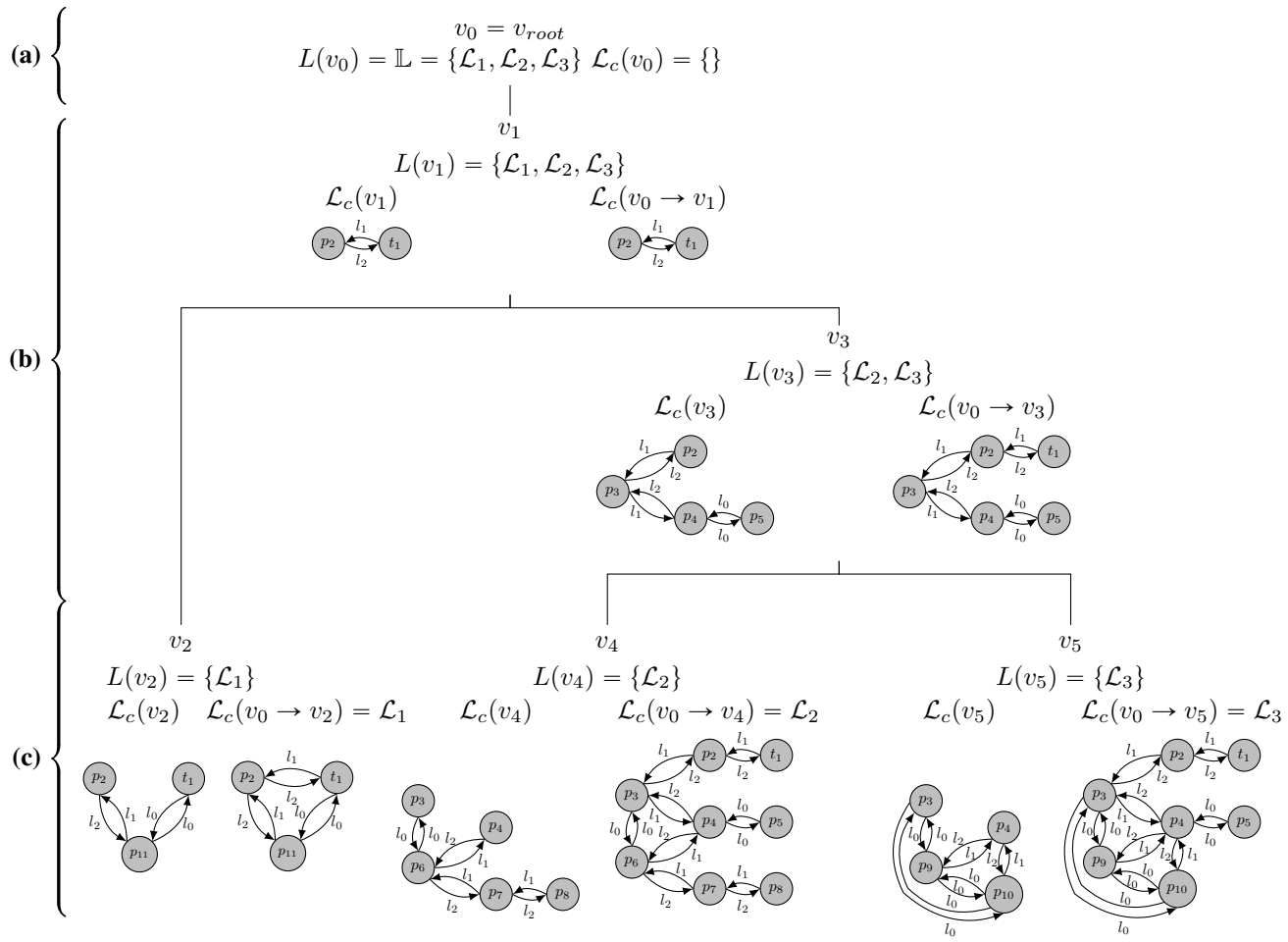


Fig. 3. An example of Basic Graph Pattern Hierarchical Clustering built (BGPHC) during the process **A.3** from the three basic graph pattern BGP_1, BGP_2 and BGP_3 of figure 2.

Algorithm 1 BuildBGPHC($\mathbb{Q}$): Φ

Input: $\mathbb{Q}$ is the set of all the known SPARQL queries
Output: Φ is the BGPHC of $\mathbb{Q}$

1: $\mathbb{BGP} \leftarrow$ ExtractBGP($\mathbb{Q}$), $\mathbb{L} \leftarrow$ BuildLinegraphs($\mathbb{BGP}$)
2: $\mathcal{L}_{c0} \leftarrow \varnothing$, $L_0 \leftarrow \mathbb{L}$, $L_0^{\complement} \leftarrow \varnothing$
▷ Create the tree with the root vertex. Each vertex is a 3-tuples.
3: $\Phi \leftarrow new\ Tree(Tuple3(\mathcal{L}_{c0}, L_0, L_0^{\complement}))$
4: $v_0 \leftarrow$ Root(Φ)
5: $C \leftarrow$ LinegraphClustering(L_0, v_0) ▷ set of clusters
6: AddChildren(v_{root}, C)
7: RecBuildBGPHC(Φ, v_0)
8: **return** Φ
9: **procedure** RecBuildBGPHC(Φ, v_p)
▷ $v_p \in V$ current position in Φ
10: **if** Children(v_p) $\neq \varnothing$ **then**
11: **for all** $v_c \in$ Children(v_p) **do**
12: **if** $|L(v_c)| > 1$ **then**
13: $C \leftarrow$ LinegraphClustering($L(v_c), v_c$)
14: AddChildren(v_c, C)
15: RecBuildBGPHC(Φ, v_c)
16: **end if**
17: **end for**
18: **end if**
19: **end procedure**

of BGPHC is unknown a priori, we use an *unsupervised* clustering. Last, we follow a *divisive* method, that is, starting with a single cluster containing all linegraphs, we recursively split the clusters until completion. Such a divisive method allows building the hierarchical clustering and the common sub-linegraphs of BGPs from the most shared to less shared.

The recursive hierarchical clustering algorithm. Algorithm 1 is a recursive clustering algorithm which finds the (non empty) minimum common arc subgraphs necessary to encompass a maximum number of linegraphs in each computed cluster. In each cluster v_i (each vertex of the cluster structure), we name $\mathcal{L}_c(v_i)$ the common sublinegraphs, that is, the subgraphs of the linegraphs sharing at least one arc.

The recursive clustering algorithm. Algorithm 2 computes the clustering. Each clustering is computed from the cluster computed at the previous level, noted v_p. During the computation of each current cluster, three data structures are built (for optimization). They respectively store, for each v_i,

- $\mathcal{L}_c(v_i)$,
- $L^{\complement}(v_i)$ the complement of $L(v_i)$ defined as follows

$$L^{\complement}(v_i) = \{\mathcal{L} \in L(v_p) \subseteq \mathbb{L} \mid \mathcal{L} \notin L(v_i)\} \quad (1)$$

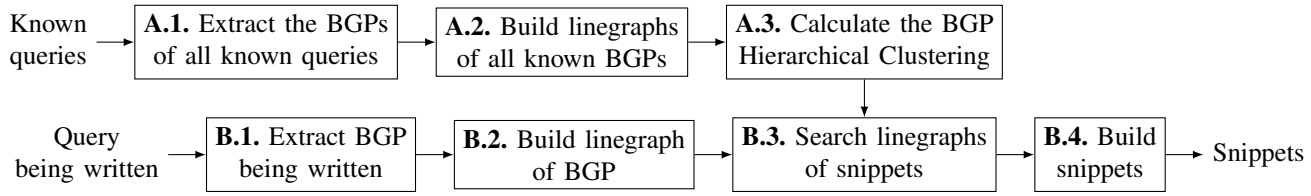


Fig. 4. Workflow of SPARQLets-Finder where **A.1-3** builds a BGPPHC and **B.1-4** calculate snippets for a query.

- $L(v_i)$, the set of linegraphs sharing at least one arc with a computed common linegraph $\mathcal{L}_c(v_i)$.

The algorithm takes in $L(v_p)$, the set of linegraphs of the cluster computed at the previous level. The algorithm first computes the minimal number of (non empty) common linegraphs, maximizing the number of clusters. Next, it finds the minimum (non empty) common arc subgraphs necessary to encompass a maximum of linegraphs in each found cluster. The clustering takes in $\mathcal{L}_c(v_0 \rightarrow v_p)$, the union of the common linegraphs computed at the precedent level, from the root vertex v_0 to the precedent cluster computed v_p:

$$\mathcal{L}_c(v_0 \rightarrow v_x) = \bigcup_{y=v_0}^{v_x} \mathcal{L}_c(v_y) \tag{2}$$

The program computing $\mathcal{L}_c(v_0 \rightarrow v_p)$ will be reused in the next section to compute snippets.

Illustration. In the BGPHC of Figure 3, the original set of linegraphs, noted $\mathbb{L}$, has three linegraphs. The starting point is represented in (a) with all linegraphs in the same initial cluster $L(v_0)$. In (b), the common subgraphs in a cluster $\mathcal{L}_c(v_i)$ have been computed and are represented. In (c), the final clusters are represented. In this example, each final cluster is composed of only one single linegraph.

We next illustrate how to use BGPHC and the LBGPD measure to suggest SPARQL snippets, using a workflow that we call SPARQLets.

D. SPARQLets-Finder

SPARQLets-Finder sequences operations in a workflow, described in Figure 4 to compute snippets to help a user write a query. We start with illustrating the process of snippets construction and explaining how they are found by BGPHC.

In B.1 of Figure 5, the first step, BGP_0 is extracted from the query being designed as input. The BGP is selected depending on the focus position in the query editor. We use an ANTLR4 parser to extract it in B.1. and also in step A.1. to extract all BGPs of the queries in the knowledge base. We chose to develop our own parser for the official SPARQL grammar [10] to parse even in presence of errors in the (partial) query submitted.

In B.2., the linegraph of the input BGP is computed, noted $\mathcal{L}_{BGP_0} = \mathcal{L}(BGP_0)$.

In B.3., Algorithm 3 on the directed tree of BGPHC is used to detect snippets. Each path between a vertex and the root constitutes a potential linegraph to build one snippet.

Algorithm 2 LinegraphClustering(L, v_p): C

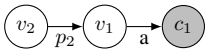
$C \leftarrow \varnothing$
▷ Compute the number of clusters
for all $\mathcal{L} \in L$ **do**
  $addInACluster \leftarrow FALSE$
  $\Delta\mathcal{L} \leftarrow \mathcal{L} \backslash \mathcal{L}_c(v_0 \rightarrow v_p)$ ▷ remove precedent $\mathcal{L}_c$
  if $(\Delta\mathcal{L} \neq \varnothing)$ **then**
    for all $\mathcal{C} \in C$ **do**
      $d_J \leftarrow \mathrm{J}_\delta(\mathcal{L}_c(\mathcal{C}), \Delta\mathcal{L})$
      if $d_J < 1$ **then**
        $\mathcal{L}'_c \leftarrow \mathcal{L}_c(\mathcal{C}) \cap \Delta\mathcal{L}$
        $L' \leftarrow L(\mathcal{C}) \cup \{\mathcal{L}\}$
        UPDATE$(\mathcal{C}, Tuple3(\mathcal{L}'_c, L', L^{\complement}(\mathcal{C})))$
        $addInACluster \leftarrow true$
      else
        $L'^{\complement} \leftarrow L^{\complement}(\mathcal{C}) \cup \{\mathcal{L}\}$
        UPDATE$(\mathcal{C}, Tuple3(\mathcal{L}_c(\mathcal{C}), L(\mathcal{C}), L'^{\complement}))$
      end if
    end for
    if not $addInACluster$ **then**
      ADDNEWCLUSTER$(C, Tuple3(\Delta\mathcal{L}, \{\mathcal{L}\}, \varnothing))$
    end if
  end if
end for
▷ Compute the definitive clusters
for all $\mathcal{L} \in L$ **do**
  $\Delta\mathcal{L} \leftarrow \mathcal{L} \backslash \mathcal{L}_c(v_0 \rightarrow v_p)$
  if $(\Delta\mathcal{L} \neq \varnothing)$ **then**
    for all $\mathcal{C} \in C$ **do**
      if $\mathcal{L} \notin (L^{\complement}(\mathcal{C}) \cup L(\mathcal{C}))$ **then**
        $d_J \leftarrow \mathrm{J}_\delta(\mathcal{L}_c(C), \Delta\mathcal{L})$
        if $d_J < 1$ **then**
          $\mathcal{L}'_c \leftarrow \mathcal{L}_c(\mathcal{C}) \cap \Delta\mathcal{L}$
          $L' \leftarrow L(\mathcal{C}) \cup \{\mathcal{L}\}$
          UPDATE$(\mathcal{C}, Tuple3(\mathcal{L}'_c, L', L^{\complement}(\mathcal{C})))$
        else
          $L'^{\complement} \leftarrow L^{\complement}(\mathcal{C}) \cup \{\mathcal{L}\}$
          UPDATE$(\mathcal{C}, Tuple3(\mathcal{L}'_c(\mathcal{C}), L'(\mathcal{C}), L'^{\complement}))$
        end if
      end if
    end for
  end if
end for
return C

In input, a SPARQL query:

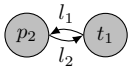
```
SELECT *
WHERE {
   ?disease wdt:P31 wd:Q12136 .
   ?drug wdt:P2175 ?disease .
}
```

B.1. Extract current basic graph pattern: BGP_0

v_1 a c_1
v_2 p_2 v_{v3}

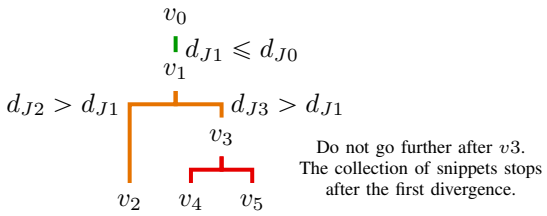


B.2. Build linegraph $\mathcal{L}_{BGP_0} = \mathcal{L}(BGP_0)$

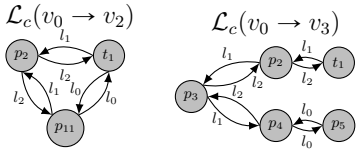


B.3. Search linegraphs of snippets

Where $d_{Ji} = Jaccard_\delta(\mathcal{L}_c(v_0 \rightarrow v_i), \mathcal{L}_{BGP_0})$



Linegraphs found:



B.4. Build snippets

Substract with $\mathcal{L}_{BGP_0}$ in input and reverse to a BGP:

$\mathcal{L}_{snippet1} = \mathcal{L}_c(v_0 \rightarrow v_2) \backslash \mathcal{L}_{BGP_0}$ $\mathcal{L}_{snippet2} = \mathcal{L}_c(v_0 \rightarrow v_3) \backslash \mathcal{L}_{BGP_0}$

$BGP_{snippet1}$
v_1 a c_1
v_1 p_{11} v_2
v_3 p_2 v_1

$BGP_{snippet2}$
v_1 p_2 v_2
v_3 p_3 v_1
v_4 p_4 v_3
v_4 p_5 v_5

Remove duplicate properties, change labels of variables and reorganize triples:

$Snippet_1$
```
?disease wdt:P18 ?image .
```
$Snippet_2$
```
?drug wdt:P2175 ?medicalConditionTreated .
?var4 p:P3354 ?positiveTherapeuticPredictorv1 .
?var4 wdt:P3433 ?biologicalVariantOf .
?positiveTherapeuticPredictorv1 ps:P3354 ?drug .
```

In output, the selected snippet by the user, here $Snippet_2$, is inserted in the initial BGP_0, resulting in BGP_{res}.
It remains for the user to improve this new query and test it:

```
?disease wdt:P31 wd:Q12136 .
?drug wdt:P2175 ?disease .
~~?drug wdt:P2175 ?medicalConditionTreated.~~
~~?var4~~ ?variant p:P3354 ?positiveTherapeuticPredictorv1.
~~?var4~~ ?variant wdt:P3433 ?biologicalVariantOf .
?positiveTherapeuticPredictorv1 ps:P3354 ?drug .
```

Fig. 5. Illustration of processes **B.1-4** to calculate snippets in function the BGPHC build with $\mathbb{Q} = \{q_1, q_2, q_3\}$ in Figure 3 and the set BGP_0 of triple patterns extracted of a query $q \notin \mathbb{Q}$.

To select these paths in the tree representation of the clustering, we use the Jaccard distance between $\mathcal{L}_{BGP_0}$ and $\mathcal{L}_c(v_0 \rightarrow v_i)$ defined in the previous section. With these distance values from the root v_0 in the BGPHC, we walk from vertex to vertex v_i as long as the similarity increases between the $\mathcal{L}_{BGP_0}$ and $\mathcal{L}_c(v_0 \rightarrow v_i)$. When the similarity decreases, the research in the tree is blocked and the destination vertices are collected. From the set of collected vertices V_s, the set of snippet linegraphs L_s is then computed (line 17 of Algorithm 3).

We select the top-five from the resulting set of snippet linegraphs based on their similarity with $\mathcal{L}_{BGP_0}$ (Line 6 of Algorithm 3). We arbitrarily limit the number of snippets to five to reduce the computation time needed to generate the snippets in B.4.

In B.4., the last step of this workflow, we convert the selected linegraphs of snippets into a list of BGPs.

At this stage, we already dispose of readable snippets. Some postprocessing is used to make them more human readable.

Algorithm 3 SearchSnippets($BGP_0, \Phi, \mathcal{K}$): S

Input: BGP_0 of a SPARQL query, Φ the BGPHC computed by the Algo 1 and $\mathcal{K}$ the knowledge on the SPARQL service of the query to build the names of the variables.
Output: S set of suggested snippets.

```
▷ J_δ(L_a, L_b) is the Jaccard distance between L_a and L_b
▷ d_Ji = J_δ(L_c(v_0 → v_i), L_BGP0) and d_Ji ∈ [0..1]
1:  L_BGP0 ← L(BGP_0)
2:  v_0 ← ROOT(Φ)                                   ▷ We start with the root
3:  L_s ← ∅                                         ▷ To start, L_s is empty
4:  d_J0 ← 1        ▷ J_δ(L_c(v_0 → v_0), L_BGP0)= J_δ(∅, L_BGP0)= 1
5:  RECSSL(L_BGP0, L_s, Φ, v_0, d_J0)
6:  L̂_s ← argmin_{L_s ∈ L'_s ⊆ L_s, |L'_s| ⩽ 5} sort(J_δ(L_s, L_BGP0))
7:  S ← BuildSnippet(L̂_s, BGP_0, K)                 ▷ Convert L to BGP
8:  return S
9:  procedure RECSSL(L_BGP0, L_s, Φ, v_p, d_J)
10:     V_childs ← CHILDREN(Φ, v_p)
11:     if V_childs ≠ ∅ then
12:         for all v_i ∈ V_childs do
13:             d_Ji ← J_δ(L_c(v_0 → v_i), L_BGP0)
14:             if d_Ji ⩽ d_J then
15:                 RECSSL(L_BGP0, L_s, Φ, v_i, d_Ji)
16:             else if d_Ji > d_J then
17:                 L_snippet ← L_c(v_0 → v_i) \ L_BGP0
18:                 L_s ← L_s ∪ {Tuple2(d_Ji, L_snippet)}
19:             end if
20:         end for
21:     else
22:         if d_J < 1 and d_J ≠ 0 then
23:             L_s ← L_s ∪ {Tuple2(d_J, L_snippet)}
24:         end if
25:     end if
26: end procedure
```

The next section presents an evaluation of this approach, both quantitatively and qualitatively (evaluation by scientists).

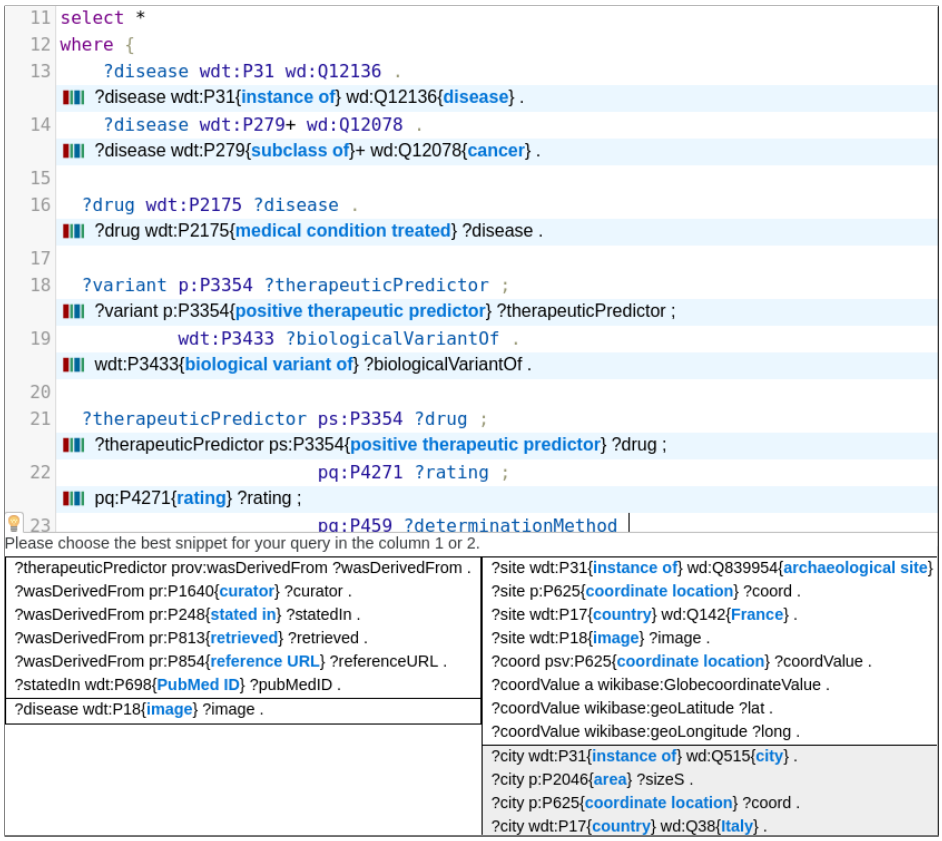


Fig. 6. We deployed two approaches in our LinkedWiki SPARQL editor to do a quantitative evaluation. During a query writing, a real user can see and chose one snippet of two different approaches: to left, SPARQLets-Finder and to right, "Full-text search".
See the editor with this query: https://io.datascience-paris-saclay.fr/exampleInsertUpdate.php?ex_id=573&action=copy

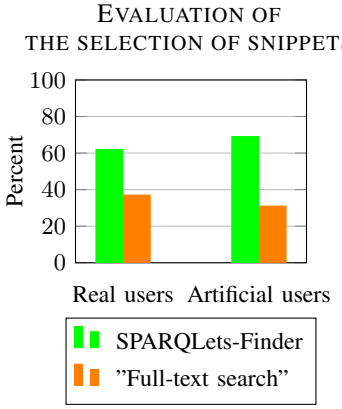


Fig. 7. Comparison between SPARQLets-Finder and "Full-text search".

V. Evaluation

In this section, we evaluate our approach both quantitatively and qualitatively. Remind that our approach aims at helping users design a query without requiring any knowledge of the ontologies involved in the query.

At the time of the evaluation, the knowledge base included 1400 BGPs extracted from 580 SPARQL queries written by humans, among which about 100 dealt with biology and biomedecine. The quantitative evaluation provides a rough measure of the effectiveness of the approach. Furthermore, the qualitative evaluation demonstrates the benefits that users can obtain from it.

A. Quantitative evaluation

We compare our technique to an alternative technique already provided by a number of database engines, notably SQL engines, under different names such as "match with query expansion" or "like" [25]. We call it here "full-text search". To capture it, in the workflow of Figure 4, we replaced Steps B.2. and B.3. by such a full-text search.

The two techniques are deployed in the SPARQL editor. This allows comparing the snippets produced by each. Figure 6 illustrates the suggestion of snippets that appears when a user clicks on the light bulb icon. Each column in the pop-up contains snippets provided by one technique. To avoid bias, the position of the column for SPARQLets-Finder is chosen randomly.

Tests with real users. At the time of the evaluation (2018/09/12) with real users (about 134 query sessions), we could measure that users selected much more often SPARQLets-Finder than "full-text search" (see Figure 7).

Synthetic tests. To confirm this trend, we considered synthetic tests. For a particular test,

- the expected result BGP_{res} is a BGP chosen randomly in the knowledge base, and
- the input BGP_0 is obtained by removing one or more triple patterns from BGP_{res} in the last position.

For comparing the snippets proposed by the two approaches to the expected results we search for the closer snippet to BGP_{res}. To do this, we use the Levenshtein similarity measure because this is the measure we tried that seemed to best capture the result expected by humans.

In the experiments, SPARQLets-Finder proved to be better in a large majority of cases. An exception is when the input BGP consists of a single triple pattern, where they both behave poorly. We intend to enrich SPARQLets-Finder to overcome the difficulty.

Results are presented in Figure 7 for 1700 tests with BGPs including at least two triple patterns. They confirm the results obtained with human users at a smaller scale. Ongoing work on quantitative evaluation includes considering several users simultaneously, assessing the ability of our approach to scale.

In the next section, we demonstrate the ability of SPARQLets-Finder to actually help scientists design their queries.

B. Qualitative

For the qualitative evaluation, we return to the running example. We report the reactions of clinicians using the system.

With only limited knowledge of the organization of the data, they are now enabled to query Wikidata, the burden of writing SPARQL queries being reduced by the autocompletion. They appreciate the support in the management of the prefixes (lines 1 to 9 on Figure 1) and the call to the the Wikidata label services (lines 32 to 34 on Figure 1). The ability to search properties and IRI using keywords also simplifies their task in constructing queries: for example, for the identification of the IRI of cancer (lines 13 to 14 on Figure 6), as well as the property *treats* between drugs and cancer (line 16 on Figure 6). Finally, the snippets turn out to be very helpful for the construction of the two most complex parts of the query: the search for positive predictors for the response (lines 18 to 23 on Figure 6), and the search for the provenance from the biomedical literature (Snippet to left in the Figure 6). In both cases, the snippets suggest structures that are already constructed and helps clinicians construct their own queries if the proposition does not already answer their need.

Figure 6 provides a snapshot where the user is provided with two snippets by SPARQLets-Finder (left-hand side) and two by the "full-text search" method. The snippets provided by SPARQLets-Finder are particularly relevant for the user.

To obtain the same advantages in other topics as well, scientists simply need to share their queries. In the experiment we presented here, only 100 queries sufficed. The effort thus remains reasonable.

VI. Conclusion

In this paper, we have considered the problem of guiding scientists in designing SPARQL queries, making it possible to exploit the full richness of the RDF linked open data. In particular, we have focused in identifying the user needs in terms of autocompletion features and we have provided solutions to these needs.

More precisely, our contribution is four-fold. First, we have collected needs on autocompletion from a large set of users and we have drawn a panorama of the current autocompletion features provided by the main SPARQL editors to answer these needs. Second, to answer the need expressed by users, we have introduced the first (to our knowledge) autocompletion approach based on *snippets*, able to provide users with completions of their query based on previous similar queries. Third, we have provided an evaluation of our approach both quantitatively, providing measures obtained on a large set of use cases, and qualitatively, based on biomedical queries. We have demonstrated that the snippets provided were particularly relevant for the user to design a new query and the effort required to learn how to use our solution remained reasonable.

Future work is planned on several directions.

First, we want to allow SPARQLets-Finder to deal with very small queries, i.e., with BGP consisting of a single triple pattern. The difficulty is that the linegraph for the BGP for a single triple pattern is meaningless. To do so, we need to introduce a new form of rankings for common sub-BGPs.

Second, we want to optimize SPARQLets-Finder. On the one hand, for each autocompletion requires several seconds to deliver a snippet, we are working in making it faster. On the other hand, we also want to introduce personalization in SPARQLets-Finder. For a particular user, we can use, for instance, her previous queries, her level of expertise, and her level of confidence to some collaborators. Information on the queries known by the system, such as the domain, could reduce the set of queries to consider and, at the same time, speed the evaluation and improve the quality of the results.

Last, we intend to extend this work to better handle federated queries. This implies considering the SPARQL services's features of BGPs and thus, the relationships between the graphs patterns of different services within the same query. Once this step will be performed, we will be able to refine the design of snippets focusing in the ontology schemas used by the services.

Acknowledgment

This work is supported by the Center for Data Science and funded by the IDEX Paris-Saclay (ANR-11-IDEX-0003-02) and by the CNRS Mastodon Project QCM-BioChem.

References

[1] K. Rafes and C. Germain, "A platform for scientific data sharing," in *BDA*, 2015.

[2] K. Rafes, S. Cohen-Boulakia, and S. Abiteboul, "Une autocomplétion générique de sparql dans un contexte multi-services," in *BDA*, 2017.

[3] M. L. Guilly, J.-M. Petit, and V.-M. Scuturici, "Sql query completion for data exploration," *arXiv preprint arXiv:1802.02872*, 2018.

[4] N. Khoussainova, Y. Kwon, M. Balazinska, and D. Suciu, "Snipsuggest: Context-aware autocompletion for sql," *Proceedings of the VLDB Endowment*, vol. 4, no. 1, pp. 22–33, 2010.

[5] S. Abiteboul, Y. Amsterdamer, T. Milo, and P. Senellart, "Auto-completion learning for xml," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 2012, pp. 669–672.

[6] J. Fan, G. Li, and L. Zhou, "Interactive sql query suggestion: Making databases user-friendly," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 351–362.

[7] Q. T. Tran, C.-Y. Chan, and S. Parthasarathy, "Query by output," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. ACM, 2009, pp. 535–548.

[8] S. Idreos, O. Papaemmanouil, and S. Chaudhuri, "Overview of data exploration techniques," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 277–281.

[9] A. Bielefeldt, J. Gonsior, and M. Krtzsch, "Practical linked data access via sparql: The case of wikidata," in *Workshop on Linked Data on the Web (LDOW)*, 2018. [Online]. Available: http://ceur-ws.org/Vol-2073/#article-03

[10] W3C SPARQL Working Group, "Recommendations of the W3C: SPARQL 1.1 (Protocol and RDF Query Language)," March 2013.

[11] L. Rietveld and R. Hoekstra, "The yasgui family of sparql clients," *Semantic Web*, vol. 8, no. 3, pp. 373–383, 2017.

[12] TSO (The Stationery Office), "Flint SPARQL editor released into semantic web community," 2011. [Online]. Available: http://www.tso.co.uk/news/2011/07/flint-sparql-editor-released-semantic-web-community

[13] DERI, NUI Galway, "Prefix.cc: namespace lookup for RDF developers," 2010. [Online]. Available: http://prefix.cc/about

[14] S. Campinas, "Live sparql auto-completion," in *Proceedings of the 2014 International Conference on Posters & Demonstrations Track-Volume 1272*. CEUR-WS. org, 2014, pp. 477–480. [Online]. Available: http://scampi.github.io/gosparqled

[15] OpenLink, "OpenLink iSPARQL," 2011. [Online]. Available: https://www.openlinksw.com/isparql

[16] N. Zaki and C. Tennakoon, "Biocarian: search engine for exploratory searches in heterogeneous biological databases," *BMC bioinformatics*, vol. 18, no. 1, p. 435, 2017.

[17] Wikimedia, "Wikidata Query," 2018. [Online]. Available: https://query.wikidata.org

[18] T. Gottron, A. Scherp, B. Krayer, and A. Peters, "Lodatio: using a schema-level index to support users infinding relevant sources of linked data," in *Proceedings of the seventh international conference on Knowledge capture*. ACM, 2013, pp. 105–108.

[19] M. Held and J. M. Buhmann, "Unsupervised on-line learning of decision trees for hierarchical data analysis," in *Advances in neural information processing systems*, 1998, pp. 514–520.

[20] D. Boley, "A scalable hierarchical algorithm for unsupervised clustering," in *Data Mining for Scientific and Engineering Applications*. Springer, 2001, pp. 383–400.

[21] S. M. Savaresi, D. L. Boley, S. Bittanti, and G. Gazzaniga, "Cluster selection in divisive clustering algorithms," in *Proceedings of the 2002 SIAM International Conference on Data Mining*. SIAM, 2002, pp. 299–314.

[22] J. Basak and R. Krishnapuram, "Interpretable hierarchical clustering by constructing an unsupervised decision tree," *IEEE transactions on knowledge and data engineering*, vol. 17, no. 1, pp. 121–132, 2005.

[23] R. Q. Dividino and G. Gröner, "Which of the following sparql queries are similar? why?" in *LD4IE@ ISWC*, 2013.

[24] W. Le, A. Kementsietsidis, S. Duan, and F. Li, "Scalable multi-query optimization for sparql," in *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*. IEEE, 2012, pp. 666–677.

[25] E. Kenler and F. Razzoli, *MariaDB Essentials*. Packt Publishing Ltd, 2015, ch. Full-Text Searches, p. 141.